CONCURRENCY: INTRODUCTION

Shivaram Venkataraman
CS 537, Spring 2020
- Project 2b is out. Due Feb 24\textsuperscript{th}, 10:00pm
- Project 1b grades very soon

Shivaram upcoming travel
- No class on Feb 27. Guest lecture March 3
- Discussion
  - No discussion Feb 20, Feb 27
  - Discussion on Tue Feb 25 at 5.30pm
AGENDA / LEARNING OUTCOMES

Virtual memory: Summary

Concurrency
  What is the motivation for concurrent execution?
  What are some of the challenges?
RECAP
SWAPPING

OS goal: Support processes when not enough physical memory
  – Single process with very large address space
  – Multiple processes with combined address spaces

User code should be independent of amount of physical memory
  – Correctness, if not performance
### Storing disk address in PTE

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
<th>present</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rw-</td>
<td>1</td>
</tr>
</tbody>
</table>

- Diagram showing the Phys Memory and Disk
- What if access vpn 0xb?
- Page is on disk
- Bring in pages from disk
- Page Fault

What if access vpn 0xb?
PAGE REPLACEMENT POLICIES

OPT: Replace page not used for longest time in future
   – Advantages: Guaranteed to minimize number of page faults
   – Disadvantages: Requires that OS predict the future; Not practical

FIFO: Replace page that has been in memory the longest
   – Advantages: Fair: All pages receive equal residency; Easy to implement
   – Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in past
   – Advantages: With locality, LRU approximates OPT
   – Disadvantages:
     • Harder to implement, must track which pages have been accessed
IMPLEMENTING LRU

Software Perfect LRU
- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU
- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement
  (especially as size of memory grows)
CLOCK ALGORITHM

Hardware
- Keep use (or reference) bit for each page frame
- When page is referenced: set use bit

Operating System
- Page Replacement:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear use bits as we search
  - Stop when find page with already cleared use bit, replace this page
CLOCK: LOOK FOR A PAGE

Physical Mem:

Use = 1, 1, 0, 1 at start

What should we evict?

Page 0 is accessed

What should we evict?

Use = x01, x0, 0, 1

Page 0: Clear bit
Page 1: Clear bit
Page 2: Selected for eviction
Page 0: Set bit

Translate Page 4

Page Fault

Evict

Add from PT
CLOCK EXTENSIONS

Replace multiple pages at once
- Intuition: Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time and track free list

Use dirty bit to give preference to dirty pages
- Intuition: More expensive to replace dirty pages
  Dirty pages must be written to disk, clean pages do not
- Replace pages that have use bit and dirty bit cleared
SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation
- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging
- Extra memory references: avoid with TLB
- Page table size: avoid with multi-level paging, inverted page tables etc.

Larger address spaces: Swapping mechanisms, policies (LRU, Clock)
CONCURRENCY
MOTIVATION FOR CONCURRENCY
Motivation

CPU Trend: Same speed, but multiple cores
Goal: Write applications that fully utilize many cores

Option 1: Build apps from many communicating processes
– Example: Chrome (process per tab)
– Communicate via pipe() or similar

Pros?
– Don’t need new abstractions; good for security

Cons?
– Cumbersome programming
– High communication overheads
– Expensive context switching (why expensive?)
CONCURRENCY: OPTION 2

New abstraction: thread

Threads are like processes, except:

multiple threads of same process share an address space

Divide large task across several cooperative threads
Communicate through shared address space
Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads

- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread

- **Defer work with background thread**
  One thread performs non-critical work in the background (when CPU idle)
CPU 1
running thread 1

CPU 2
running thread 2

RAM

What state do threads share?

Thread ID - Different
Code and Heap segments - Same
Instruction ptr & other reg - Different
Stack for local variables - Different

Code

heap

stack $T1$

"limit _n"

stack $T2$
Multiple threads within a single process share:
- Process ID (PID)
- Address space: Code (instructions), Most data (heap)
- Open file descriptors
- Current working directory
- User and group id

Each thread has its own
- Thread ID (TID)
- Set of registers, including Program counter and Stack pointer
- Stack for local variables and return addresses
  (in same address space)
User-level threads: Many-to-one thread mapping
- Implemented by user-level runtime libraries
  Create, schedule, synchronize threads at user-level
- OS is not aware of user-level threads
  OS thinks each process contains only a single thread of control

Advantages
- Does not require OS support; Portable
- Lower overhead thread operations since no system call

Disadvantages?
- Cannot leverage multiprocessors
- Entire process blocks when one thread blocks
OS SUPPORT: APPROACH 2

Kernel-level threads: One-to-one thread mapping
- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages
- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages
- Higher overhead for thread operations
- OS must scale well with increasing number of threads
```c
volatile int balance = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        balance++;
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

```
> ./threads 100000
Initial value : 0
Final value   : 162901
```

Expect: 200,000 or other number
balance = balance + 1;
balance at 0x9000

State:
0x9000: 100
%eax: 0x195
%rip = 0x195

0x195 mov 0x9000, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9000

thread control blocks:

Thread 1
%eax: 100 101
%rip:

Thread 2
%eax: 102 102
%rip:


1. Run 0x195
2. Inc 0x19a
3. Save 0x19d

1. Run 0x195
2. Inc
3. Store
balance = balance + 1;
balance at 0x9cd4

**State:**
0x9000: 100
%eax:
%rip = 0x195

0x195 mov 0x9000, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9000

Context Switch in opportune moment
**Thread 1**

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

**Thread 2**

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```
Process A with threads TA1 and TA2 and process B with a thread TB1.

1. With respect to TA1 and TA2 which of the following are true?
   - They have their own TID
   - They have their own Program Counters

2. Which of the following are true with respect to TA1 and TB1?
   - They have their own code, heap, stack
   - They have their own page tables
Concurrency leads to non-deterministic results
  – Different results even with same inputs
  – race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!
WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group
That is, we want them to be atomic

```
  mov 0x123, %eax
  add %0x1, %eax
  mov %eax, 0x123
```

More general: Need mutual exclusion for critical sections
if thread A is in critical section C, thread B isn’t
(okay if other threads do unrelated work)
Synchronization

Build higher-level synchronization primitives in OS
Operations that ensure correct ordering of instructions across threads
Use help from hardware

Motivation: Build them once and get them right

- Monitors
- Locks
- Semaphores
- Condition Variables
- Loads
- Stores
- Test&Set
- Disable Interrupts
LOCKS
Goal: Provide mutual exclusion (mutex)

Allocate and Initialize
   - `Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;`

Acquire
   - Acquire exclusion access to lock;
   - Wait if lock is not available (some other process in critical section)
   - Spin or block (relinquish CPU) while waiting
   - `Pthread_mutex_lock(&mylock);`

Release
   - Release exclusive access to lock; let another process enter critical section
   - `Pthread_mutex_unlock(&mylock);`
LOCK IMPLEMENTATION GOALS

Correctness
- Mutual exclusion
  Only one thread in critical section at a time
- Progress (deadlock-free)
  If several simultaneous requests, must allow one to proceed
- Bounded (starvation-free)
  Must eventually allow each waiting thread to enter

- Fairness: Each thread waits for same amount of time
- Performance: CPU is not used unnecessarily
IMPLEMENTING SYNCHRONIZATION

Atomic operation: No other instructions can be interleaved

Approaches
- Disable interrupts
- Locks using loads/stores
- Using special hardware instructions
IMPLEMENTING LOCKS: W/ INTERRUPTS

Turn off interrupts for critical sections
- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```c
void acquire(lockT *l) {
    disableInterrupts();
}

void release(lockT *l) {
    enableInterrupts();
}
```

Disadvantages?
- Only works on uniprocessors
- Process can keep control of CPU for arbitrary length
- Cannot perform other necessary work
Does this work? What situation can cause this to not work?
Race Condition with Load and Store

*lock == 0 initially

Thread 1

\[ \text{while}(*\text{lock} == 1) \]

\[ *\text{lock} = 1 \]

Thread 2

\[ \text{while}(*\text{lock} == 1) \]

\[ \text{lock} = 0 \]

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic
NEXT STEPS

Project 2b: Out now

Next class: More about locks!

Reminder:
  No discussion today!
  Next discussion on Tue